An Adaptive Markov Model for the Timing Analysis of Probabilistic Caches

CHAO CHEN, Polytechnique Montréal
GIOVANNI BELTRAME, Polytechnique Montréal

Accurate timing prediction for real-time embedded software execution is becoming a problem due to the increasing complexity of computer architecture, and the presence of mixed-criticality workloads. Probabilistic caches were proposed to set bounds to Worst Case Execution Time (WCET) estimates and help designers improve real-time embedded system resource use. Static Probabilistic Timing Analysis (SPTA) for probabilistic caches is nevertheless difficult to perform, because cache accesses depend on execution history, and the computational complexity of SPTA makes it intractable for calculation as the number of accesses increases. In this paper, we explore and improve SPTA for caches with evict-on-miss random replacement policy using a state space modeling technique. A non-homogeneous Markov model is employed for single-path programs in discrete-time finite state space representation. To make this Markov model tractable, we limit the number of states and use an adaptive method for state modification. Experiments show that compared to the state-of-the-art methodology, the proposed adaptive Markov chain approach provides better results at the occurrence probability of 10^{-15}: in terms of accuracy, the state-of-the-art SPTA results are more conservative, by 11% more on average. In terms of computation time, our approach is not significantly different from the state-of-the-art SPTA.

CCS Concepts: •**Theory of computation** → **Probabilistic computation; Design and analysis of algorithms;**

Additional Key Words and Phrases: Probabilistic, real-time systems, cache

ACM Reference Format:
Chao Chen and Giovanni Beltrame, 2017. An Adaptive Markov Model for the Timing Analysis of Probabilistic Caches. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 0 (0), 25 pages.
DOI: 0000001.0000001

1. INTRODUCTION

A time-critical embedded computing system, such as a satellite on-board computer, requires accurate timing prediction of software execution. If events are not managed within a certain timeframe, the result may be catastrophic. Historically, these systems were kept at a minimum of complexity to minimize the occurrence of failures and to maintain high timing predictability. However, to address the increasing complexity of applications and their corresponding need for performance, more advanced architectures using multi-stage pipelines, several memory hierarchy levels and even Multi-Processor System-on-Chip (MPSoC) designs [Martin 2006] are proposed.

These traditional deterministic computer architectures make software timing behavior almost impossible to accurately predict. Normally, the execution time of an application on a deterministic architecture follows a distribution that might have some corner cases which are beyond normal operation. A conservative estimation will place the Worst Case Execution Time (WCET) far away from the actual maximum time used

Author's addresses: C. Chen, G. Beltrame, Département de Génie Informatique et Génie Logiciel, Polytechnique Montréal, Montréal, Canada.

DOI: 0000001.0000001

by the application [Bernat et al. 2002], especially when considering possible interactions with other tasks. This would lead to a large overestimation of the computing resources needed for the task [Cazorla et al. 2013].

To help predict timing behavior, probabilistic real-time systems were introduced. Such systems have very low pathological occurrence probabilities, which are hard to test and predict. [Quinones et al. 2009] study an instruction cache with randomized replacement (random replacement pre-existed their work), showing that it provides tighter bounds for pathological cases in which systematic cache misses happen and their probabilistic WCET (pWCET) can be empirically derived by experiments. Moreover, some commercial real-time systems have adopted time-randomized caches as well, such as the ARM processor with a pseudo-random cache replacement policy[1].

Two timing analysis techniques are proposed in literature [Wilhelm et al. 2008]: measurement based timing analysis and static timing analysis, which have their probabilistic counterparts in Measurement Based Probabilistic Timing Analysis (MBPTA) and Static Probabilistic Timing Analysis (SPTA) respectively. The result of SPTA and MBPTA is expressed in terms of pWCET, i.e. an exceedance function that shows the probability of an application to exceed a given execution time.

MBPTA is an empirical method: it is based on repeated testing of an application to estimate its timing probability distribution. Generally, MBPTA requires a large amount of data from simulations or testing on real systems to get accurate results. [Cucu-Grosjean et al. 2012] propose an MBPTA methodology based on Extreme Value Theory (EVT) [De Haan and Ferreira 2007; Beirlant et al. 2006], which needs only a few hundred runs for MBPTA.

SPTA uses a different approach: it is based on detailed knowledge of software and hardware. Together with simulation models and theoretical analysis, a precise timing analysis or a timing bound can be obtained. For caches, several variables are used for the bound calculation, e.g. *reuse distance* and *cache associativity*. *Reuse distance* defines the degree of separation between two accesses to the same memory address. pWCET estimates can be computed with the help of *reuse distance* and *cache associativity*.

In this paper, we present a methodology for SPTA for set-associative instruction and data caches with random replacement policy, which is based on the Markov chain model in [Chen et al. 2016]. It takes single-path programs as inputs and computes exceedance probabilities with respect to execution time (the number of processor cycles in our simulations). The calculation is performed using state space techniques, and it is based on a non-homogeneous Markov chain model [Serfozo 2009]. At every step, the current status of the system can be represented as a state vector with a corresponding probability vector, and the transition matrix for next step is calculated accordingly. To perform timing analysis, timing distribution vectors–which are used for timing representation and analysis–are assigned for each state. We implement another precise SPTA methodology [Altmeyer et al. 2015] that can also be used for accurate timing analysis, and find out that it provides WCET bounds that are 11% looser on average in terms of geometric mean than the results from the proposed method, while having a similar computational cost. With the proposed Markov chain based method, we can evaluate cache impacts on system performance, which helps the design of real-time embedded systems.

The rest of the paper is organized as follows: related work is discussed in Section 2; system modeling is explained in Section 3; timing analysis for the system is demonstrated in Section 4; an adaptive method is introduced in Section 5 to limit the com-

[1] http://infocenter.arm.com/help/index.jsp/

putational complexity of the Markov chain based model; real-world benchmarks are evaluated in Section 7; and finally Section 8 draws come concluding remarks.

2. RELATED WORK

There have been few research efforts on timing analysis for probabilistic systems. [Bernat et al. 2002] develop a WCET analysis method for probabilistic hard real-time systems, in which the concept of a probabilistic system–whose execution deadline must be met by given probabilities–is introduced. They use the notion of execution profiles for timing representation and analysis. To help determine the WCET of programs, [Bernat et al. 2005] propose an approach based on *copulas*. It uses the dependence structure description of programs for computing the WCET, and when unavailable, it provides a lower bound.

Traditional computing systems are deterministic, and their timing analysis depend on execution history, whose computational complexity increases exponentially as the program executes. To reduce the dependency on execution history, probabilistic systems are introduced, implemented in hardware and software.

By using hardware techniques, programmers do not need to modify software and the system WCET can be improved by hardware modifications at architectural level. The method to realize this is to modify the behavior of the cache–which is a bridge between processor and main memory–and make it random. [Mezzetti et al. 2015] show that time-randomized caches bring several benefits to hard RT system: it reduces user's efforts for timing analysis and provides tight WCET.

The behavior of a cache is determined by two policies: replacement policy and placement policy, and they are made random respectively for pWCET analysis.

For cache replacement, every time a new memory request comes into the cache set from the main memory, one cache block in this set will be selected and evicted. The new address is put into the position of the evicted block. There are several replacement policies for conventional caches, such as First-In-First-Out (FIFO), Least Recently Used (LRU), Most Recently Used (MRU) [Al-Zoubi et al. 2004], etc. To make the cache behavior random for a new memory address, one can adopt random replacement policy . When using a random replacement policy, every time the cache eviction happens, a cache block is selected randomly to be replaced by the new memory request. [Quinones et al. 2009] study a random replacement policy for standard and skew-associate caches and they compare simulation results with caches using the LRU replacement policy, because it performs best in terms of predictability [Reineke et al. 2007]. The authors show that caches with random replacement policy reduce performance anomalies. For example, in one case study, the hit ratio of a cache using the LRU is from 0.41 to 0.93; while for a cache with random replacement policy, it varies from 0.64 to 0.94.

The cache placement policy has an impact on cache behavior as well. For cache placement, when choosing the cache set, a conventional cache uses several bits of the memory address. [Schlansker et al. 1993] propose a random placement policy and investigate its impact by matrix operations. This policy distributes cache entries more uniformly, and cache miss ratio is lower than that from conventional caches. [Topham and Gonzalez 1999] use a random placement policy and the results show that it can reduce cache conflicts and improves system performance. However, the random placement policy in [Schlansker et al. 1993; Topham and Gonzalez 1999] adopts a pseudo-random hash function that depends on memory addresses. Hence for a given memory layout, it always produces the same placement distribution. [Kosmidis et al. 2013a] propose a cache with a random replacement policy and a parametric random placement policy, which requires little overhead in terms of complexity and energy consumption. The introduction of the parameter into the hash function ensures that the placement dis-

tribution is randomized for the same memory layout, so that it is feasible to apply probabilistic timing analysis.

In addition to aforementioned hardware techniques, software techniques (e.g. compiler and runtime techniques) can also be applied to make a system behavior random. [Berger and Zorn 2006] present DieHard, a runtime system, to allocate memory randomly. The *probabilistic memory safety* is achieved by using a large heap space. The DieHard manager deals with objects in the heap and reduces memory error probabilities. [Kosmidis et al. 2013b] propose a software approach to randomize the behavior of conventional caches for use with probabilistic timing analysis. This work modifies code and data memory objects offline using compiler and linker. When the program starts or objects are allocated, the memory objects are placed in random locations by dynamic randomization code in the executable. Dynamic randomization challenges safety requirements (e.g. ISO26262) in the automotive domain. To solve this issue, [Kosmidis et al. 2014b] present a static software randomization method. Several binary files are produced for the same program, in which memory objects are created with offsets to achieve random effects. [Kosmidis et al. 2016] develop a software tool to modify source code of the program, which realizes randomization without modifying existing toolchains.

In pWCET analysis, the result is expressed in terms of a density function or an exceedance function: it shows the probability of an application for given execution or the probability to exceed a given execution time. This can further be classified into three categories: Measurement-Based Probabilistic Timing Analysis (MBPTA), Static Probabilistic Timing Analysis (SPTA) and the combination of both methods.

In MBPTA, execution time measurements are collected and predictions are made using Extreme Value Theory (EVT). EVT [De Haan and Ferreira 2007; Beirlant et al. 2006] is a statistical methodology that studies extremely rare events (i.e. events at the tails of the distribution) that may have severe consequences, when little experimental evidence is available. Usually two methods can applied to EVT: Block Maxima (BM) [Cucu-Grosjean et al. 2012] and Peaks Over Threshold (POT) [Bernardara et al. 2014].

In [Burns and Edgar 2000; Edgar and Burns 2001], Burns and Edgar demonstrate how to predict execution time with measurements by an EVT method. Raw data are fit to the Gumbel distribution [Gumbel 2012] and results are represented as a density function with respect to execution time. [Hansen et al. 2009] explain why raw data fitting in [Edgar and Burns 2001] is incorrect. A BM method using EVT for WCET distribution estimation is thus presented. [Griffin and Burns 2010] investigate assumptions required by the Gumbel distribution, and study precision sacrificed due to this statistical method. Additional restrictions on the EVT method are proposed for safe applications. [Lu et al. 2011] propose a new way of sampling mechanism to estimate program execution time on a single processor and EVT is combined with this sampling technique. A more recent work using EVT is from [Cucu-Grosjean et al. 2012]. A BM method is applied to fit the Gumbel distribution using a quantile plot, needing only a few hundred simulation runs. This significantly reduces the number of required measurements. [Kosmidis et al. 2014a] study how processor architectures should be modified to meet MBPTA requirements. [Wartel et al. 2013] apply MBPTA to real avionics applications and results show tight pWCET estimates. [Abella et al. 2014b] investigate when MBPTA fails due to pathological cases and propose *Heart of Gold* techniques to detect these cases. [Lesage et al. 2015b] introduce a framework for MBPTA result evaluation. Synthetic tasks are used to provide realistic data and actual WCET can be computed using proposed framework.

Apart from statistical analysis of measurements, another way of PTA is Static PTA (SPTA). This requires detailed knowledge of software and hardware. A timing bound can be obtained by theoretical analysis: with the given assumptions, the SPTA method

analyzes instruction or data caches and obtains a probabilistic distribution of the programs execution time.

Several works on SPTA have been proposed for caches with random replacement policy. [Zhou 2010] proposes a cache hit formula using reuse distance–the number of memory addresses accessed between two consecutive references to the same memory address–which simplifies computational complexity significantly. The probabilities for each cache access are made independent, and the final result is the convolution of all cache accesses. However, [Cazorla et al. 2013; Altmeyer and Davis 2014] have found his methodology unsound. [Quinones et al. 2009; Kosmidis et al. 2013a] give other formulea for evict-on-miss caches, and [Cucu-Grosjean et al. 2012; Cazorla et al. 2013] perform evict-on-access timing analysis using these formulea. However, [Kosmidis et al. 2013a] may overestimate the cache hit ratio [Davis 2013]. Thus, the result of probabilities for timing may be too optimistic and incorrect in this case.

[Davis et al. 2013] develop a formula using reuse distance only for evict-on-miss caches, and [Altmeyer and Davis 2014] prove it to be optimal when only reuse distance is known. Multi-path programs are also analyzed by assuming that they are bounded. Besides, pre-emption impacts are taken into account for timing analysis. [Altmeyer and Davis 2014] have proposed an exhaustive analysis approach for SPTA. To reduce the computational complexity, the exhaustive approach can be combined with simplified formulae. This approach is improved in [Altmeyer et al. 2015], with an improved algorithm for SPTA. [Griffin et al. 2014] propose a methodology from the field of Lossy Compression and they use a fully-associative cache for timing analysis throughout their work. By using *May* and *Must* Analysis, the result is more accurate with appropriate parameters. To demonstrate the impact of time-randomized caches, [Reineke 2014; Abella et al. 2014a; Altmeyer et al. 2015] have done comparisons between caches using LRU and random replacement policy. [Lesage et al. 2015a] develop an SPTA for multi-path programs. A worst-case execution path is obtained using a joint function by exploring cache states and path inclusions. Based on SPTA from [Altmeyer and Davis 2014], the pWCET can be calculated.

Hybrid analysis combines both MBPTA and SPTA for timing behavior prediction. So far, very little research has been done for hybrid timing analysis. [Bernat et al. 2002; Bernat et al. 2003] introduce a hybrid timing analysis for probabilistic hard RT systems: an RT program is analyzed and its structure is represented as a syntax tree. Instrumentation and trace are generated, so that distributions of all blocks can be produced locally using SPTA or MBPTA. A traversal of the syntax tree is used to calculate the WCET of the program. Due to dependencies between different blocks, copulas is proposed by [Bernat et al. 2005] to obtain a lower bound.

In this paper, we propose an adaptive Markov chain based SPTA methodology for time-randomized caches. Rather than using reuse distance only, this method adopts more information and produces more accurate results. By limiting number of states used in the Markov chain model, the computational complexity has been restrained to make the calculation feasible.

3. SYSTEM MODEL

In this section, we present a methodology to model the timing of a system with a probabilistic cache using state space model. The next state of the system with random replacement caches solely depends on current state, which satisfies the Markov property and can be represented as a Markov chain. A set-associative cache is used as an example, but note that a direct mapped cache can be seen as a special case of set-associative cache, in which the associativity equals 1; a fully-associative cache is another special case, in which the associativity equals the number of available cache blocks.

3.1. Cache Architecture

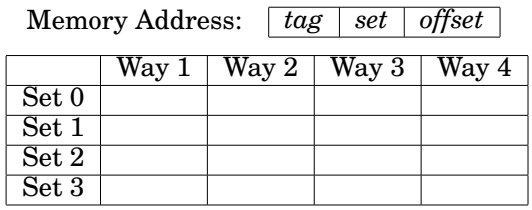


Fig. 1: Set-associative cache representation

A set-associative cache is shown in Figure 1. This cache has several sets, and for each of those it provides a number of ways to store cache blocks. Each memory address used by the cache is divided in three parts: *tag* bits, *set* bits and *offset* bits. *offset* bits locate the data within each cache block, *set* bits are used to find which cache block should be selected for a given memory address. As multiple memory addresses can be stored in the same cache block, *tag* bits are stored within each cache block for comparison to identify the correct memory address it refers to. There exist several cache policies that describe how addresses are placed and replaced in the cache. In this paper, we consider a cache with modulo placement policy and evict-on-miss random replacement policy.

For the modulo placement policy, the *set* bits are used to select the cache set in which the data will be stored using the modulo operation. With an evict-on-miss random replacement policy, every time a cache miss happens, a way is selected randomly, and the cache block data are replaced by the new memory content.

To calculate the timing distribution of a probabilistic system, we first obtain the timing distribution of each cache set and the timing associated with the whole cache can be obtained by performing a convolution across all sets. Since modulo placement policy is adopted, memory addresses in different cache sets are stored separately and do not affect each other, i.e. the memory address in one cache set does not change the hit or miss probabilities in another cache set. Hence they are independent of each other statistically. As a result, the final timing distribution can be obtained using convolutions.

3.2. State Space Exploration

Let us assume there are distinct memory addresses $M = \{a, b, c, \cdots\}$ that are allocated to one cache set. The state space $\mathbb{S}$ can be constructed in a way such that the combinations of the distinct memory addresses are elements of the state space. The element $s_i \in \mathbb{S}$ is the state of the system, and it represents a unique cache configuration, i.e. the memory address layout of the system. The state space $\mathbb{S}$ is formally defined as:

$$\forall A \subseteq M, A \in \mathbb{S}$$

Let N_w be the number of ways (i.e. associativity) of the cache set. Then we have $|s_i| \leq min(N_w, |M|)$, i.e. the number of distinct memory addresses in a state is less than or equal to the minimal value between cache associativity and the number of distinct memory addresses for this set.

Figure 2 is used as an illustration of state space construction, in which we define states $\mathbb{S} = \{s_0, s_1, s_2, s_3, \cdots\}$ that correspond to the following configurations:
$s_0 = \emptyset$: empty cache
$s_1 = \{a\}$: address a in cache
$s_2 = \{b\}$: address b in cache

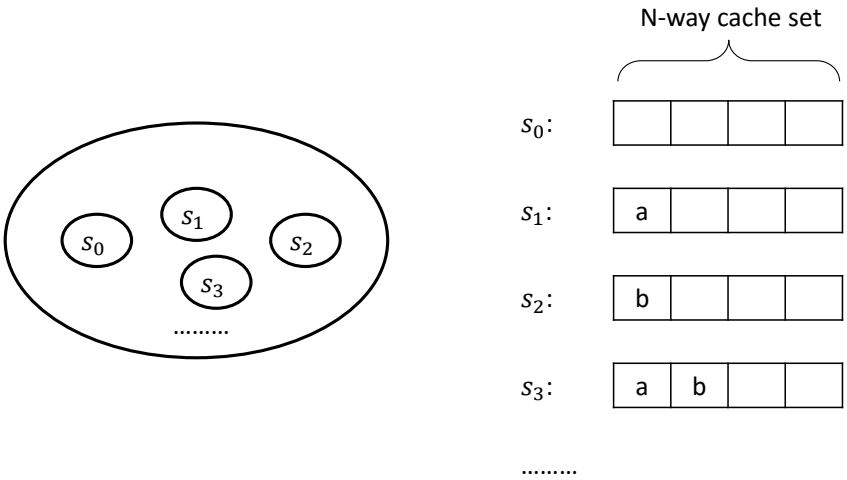


Fig. 2: State space exploration

$s_3 = \{a, b\}$: address a, b in cache

$\vdots$

A cache memory is generally organized in blocks of more than one byte. In the following analysis the term address refers to the block identifier, i.e. the tag address.

Given this state representation, one set of a time-randomized cache can be modeled with the following Markov chain:

$$S_n = S_{n-1} \cdot P_{n-1} \tag{1}$$

where S represents state occurrence probability vector for the cache and P represents the transition matrix which determines how the current state probability S is transformed into a new state S. S_n and P_n represent the state probability vector and transition matrix at step n. We assume that initially there are no memory addresses in the cache, i.e. the system starts executing with an empty cache. Let us suppose that there are N states, i.e. $|\mathbb{S}| = N$, then the state probability vector is

$$S = [Pr(s_0), Pr(s_1), \cdots, Pr(s_{N-1})] \tag{2}$$

where $Pr(s_i)$ is the probability of state s_i.

The number of states N in S_n is

$$N = \sum_{k=0}^{l} \left(\begin{array}{c} |M| \\ k \end{array} \right) \tag{3}$$

with $|M|$ the number of addresses associated with the set, and $l = min(N_w, |M|)$ i.e. the minimal value between cache associativity and $|M|$.

From Equation (3), we can see that the number of states N is a function of cache associativity N_w and the number of addresses $|M|$ for one set. For a specific cache, as the number of memory addresses $|M|$ increases, the computational complexity increases polynomially with a large exponent $|M|$, which becomes eventually intractable for computation. However, the cache organization in blocks helps reducing the total number of memory addresses. Besides, as cache size increases, the probability for all code and data to reside in one cache set decreases, which effectively lowers the value of variable $|M|$.

3.3. Transition Matrix Calculation

Every time a new memory address is accessed, the cache state may change, and the transition matrix for next step varies accordingly. Therefore, we need a non-homogeneous Markov chain model, i.e. a Markov chain model whose transition matrix varies over time. The way to compute the transition matrix at each step is demonstrated in this section.

The transition matrix is represented as

$$P = \begin{pmatrix} p_{0\to0}, & p_{0\to1}, & \cdots, & p_{0\to N-1} \\ p_{1\to0}, & p_{1\to1}, & \cdots, & p_{1\to N-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{N-1\to0}, & p_{N-1\to1}, & \cdots, & p_{N-1\to N-1} \end{pmatrix} \tag{4}$$

where $p_{i\to j}$ is the probability for the system to go from state s_i to state s_j. In our model, the probability $p_{i\to j}$ varies constantly depending on current system state and the transition matrix thus needs to be computed at each time step.

ALGORITHM 1: Transition matrix calculation

Data: State s_i, memory address a
Result: Transition matrix element $p_{i\to j}$

```
1  N_w ← cache associativity;
2  if a ∈ s_i then
3  |   p_{i→i} ←1; //cache hit
4  end
5  if a ∉ s_i then
6  |   //cache miss
7  |   for b ∈ s_i and s_j=s_i \ {b} ∪ {a} do
8  |   |   p_{i→j} ←1/N_w; //one address is replaced
9  |   end
10 |   if |s_i| < N_w then
11 |   |   for s_j=s_i ∪ {a} do
12 |   |   |   p_{i→j} ←(N_w − |s_i|)/N_w; //one address is added
13 |   |   end
14 |   end
15 end
```

Algorithm 1 shows how to compute the transition matrix. It takes two inputs (state s_i and the incoming memory address a) and produces one output (transition matrix element $p_{i\to j}$). The algorithm checks state s_i and generates the transition matrix elements accordingly as follows:

Line 2:. If the requested memory address is in the state ($a \in s_i$), there is a cache hit. In this case, the cache will not change its state and it thus has a probability of 1, i.e. $p_{i\to i} = 1$.

Line 5:. If the requested memory address is not in the state ($a \notin s_i$), there is a cache miss and the transition matrix is computed. This is the most complex case: the new memory address may replace an existing cache block, or it may be put into a new cache block and probabilities have to be computed accordingly. In our target cache, the probability of replacing an existing cache block is $1/N_w$ (see Line 8), where N_w is the cache associativity. This is because we consider an evict-on-miss time-randomized cache, and a cache block is randomly selected for replacement with probability $1/N_w$. The probability for a memory address to be placed in an empty

cache block is $(N_w - |s_i|)/N_w$ (see Line 12), where $|s_i|$ is the number of blocks in use for the current state s_i. This is due to the fact that if the new memory address does not cause a replacement, it can only be put into an empty cache block. The number of empty cache blocks is $N_w - |s_i|$, and they are chosen from N_w ways. Therefore the probability is $(N_w - |s_i|)/N_w$. Example 3.1 is given for illustration.

Example 3.1. Suppose we have a cache set with associativity of 4. The first and second cache blocks have been used, and the third and fourth cache blocks are empty. When a new memory address is cached, the probability of going into the first cache block is $1/N_w$, i.e. 1/4. Similarly, for the second cache block, the probability is 1/4. They are computed separately, because they have different addresses, which represent different states s_i. The probability of loading a block into an empty cache blocks is $(N_w - |s_i|)/N_w$, i.e. 1/2. Since the third and fourth cache blocks are both empty, loading into one or the other has the same effect, and therefore it represents a single state considering the probability of both.

Using Algorithm 1, Equation (1) can be used to describe state transitions of the system, provided the initial distribution S_0 is known. However, the cumulative timing information, which shows timing with respect to probability, is still unknown. To solve this issue, Section 4 introduces the vector that stores timing information for each state.

4. TIMING ANALYSIS

To describe the timing behavior of a system, we employ a vector containing timing information. This vector–together with the state space model described in Section 3–can describe the system timing behavior, where the state space model specifies occurrence probabilities of all states, and the timing vector specifies how the execution time is distributed for each state.

4.1. Timing Representation

A vector C_i can be used to denote the timing distribution in terms of number of cycles for a state s_i, and a vector CP_i can represent the probability of occurrence for C_i. Note that the number of cycles is different from the time step used in the Markov chain. At each time step, one memory address is accessed and different number of cycles may be applied to the timing analysis according to the system status (e.g. 1 cycle for a cache hit and 100 cycles for a cache miss). Then we have

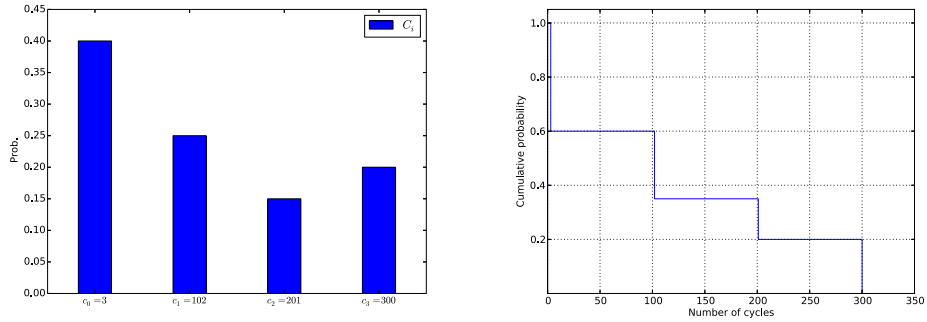
$$C_i = [c_0^i, c_1^i, \cdots]$$

$$CP_i = [Pr(c_0^i), Pr(c_1^i), \cdots]$$

where c_j^i represents the program duration in cycles in ascending order for state s_i and $Pr(c_j^i)$ the occurrence probability for c_j^i. As an example, Figure 3a shows the timing distribution in C_i and CP_i for state s_i. One can see that the probabilities for a duration of 3, 102, 201 and 300 are 0.40, 0.25, 0.15 and 0.20 respectively.

4.2. SPTA Convolution

For set-associative caches, we use convolutions to get the timing of different cache sets. In probability theory and statistics, if two random variables are independent, then the sum of them follows a distribution which is the convolution of both distributions. By using modulo placement policy, we make sure that memory addresses in one cache set do not affect hit or miss probabilities of memory addresses in another cache set. Therefore independence is guaranteed. We show the convolution in detail by using Execution Time Profile (ETP).



(a) Timing distribution (b) Cumulative timing

Fig. 3: Timing analysis example

The Execution Time Profile (ETP) is used to represent timing information and its associated probability. The ETP for state s_i is defined as

$$ETP_i = \{C_i, CP_i\} = \{[c_0^i, c_1^i, \cdots], [Pr(c_0^i), Pr(c_1^i), \cdots]\}$$

where C_i is the timing distribution vector, and CP_i is its corresponding occurrence probability vector.

Suppose there are two ETPs: ETP_i and ETP_j, and their convolutions is ETP_k. Then the convolution is performed as follows (the symbol $*$ is used as the convolution operator)

$$\begin{aligned} ETP_k &= ETP_i * ETP_j = \{C_i, CP_i\} * \{C_j, CP_j\} \\ &= \{[c_0^i, c_1^i, \cdots], [Pr(c_0^i), Pr(c_1^i), \cdots]\} * \{[c_0^j, c_1^j, \cdots], [Pr(c_0^j), Pr(c_1^j), \cdots]\} \\ &= \{C_k, CP_k\} = \{[c_0^k, c_1^k, \cdots], [Pr(c_0^k), Pr(c_1^k), \cdots]\} \end{aligned}$$

where

$$c_l^k = c_m^i + c_n^j \tag{5}$$

$$Pr(c_l^k) = \sum_{c_l^k = c_m^i + c_n^j} Pr(c_m^i) Pr(c_n^j) \tag{6}$$

The convolution of two ETPs is demonstrated in Example 4.1. Two ETPs–ETP_1 and ETP_2–are provided. ETP_1 has three possible timing values: $[1, 2, 3]$ and their probabilities are $[0.1, 0.3, 0.6]$; the timing distribution and corresponding probabilities for ETP_2 are $[1, 3]$ and $[0.2, 0.8]$ respectively. From Equation (5), we can see the element in the new timing distribution is the sum of timing distributions of elements in ETP_1 and ETP_2, i.e. $[2, 3, 4, 5, 6]$. The new probability vector is calculated using Equation (6). Its element is the sum of product of two elements in probability vectors of ETP_1 and ETP_2, provided the sum of corresponding timing distributions are the same. For example, for the cycle 4, its corresponding probability is 0.2, which consists two parts: the first part is the sum of cycle 1 in ETP_1 and cycle 3 in ETP_2. The second part is the sum of cycle 3 in ETP_1 and cycle 1 in ETP_2. So the corresponding probability is $0.1 \times 0.8 + 0.6 \times 0.2 = 0.2$.

Example 4.1.

$$ETP_1 = \{[1,2,3],[0.1,0.3,0.6]\}, ETP_2 = \{[1,3],[0.2,0.8]\}$$

$$\begin{aligned} ETP_1 * ETP_2 &= \{[1,2,3],[0.1,0.3,0.6]\} * \{[1,3],[0.2,0.8]\} \\ &= \{[2,3,4,5,6],[0.02,0.06,0.2,0.24,0.48]\} \end{aligned}$$

4.3. Cumulative Timing

Having the timing information for each cache set, we can compute a cumulative timing plot. This gives us an exceedance function, showing the probability of exceeding a certain program duration in cycles. The timing distribution is the same as from the timing vector, i.e. C_i. The cumulative probability is denoted as CPC_i, and its jth element is calculated as follows

$$CPC_i[j] = \sum_{CP_i[k]>CP_i[j]} CP_i[k] = \sum_{Pr(c_k^i)>Pr(c_k^j)} Pr(c_k^i) \tag{7}$$

where n is the number of elements in CP_i.

As an example, the cumulative probabilities to exceed a program duration of 3, 102, 201 and 300 are 0.6, 0.35, 0.2 and 0 respectively. The result is plotted in Figure 3b.

4.4. Timing Integration

From previous sections, it can be seen that timing vectors can be used to express the timing behavior of a system. In this section, we discuss how to integrate timing vectors into our state space model based on a non-homogeneous Markov chain model.

Since we use a Markov chain model for our system, at every step the system can be described by the states in the state space. To calculate timing information, timing vectors are integrated into the Markov chain model. Each state s_i is assigned a vector C_i to keep its timing. The timing vector may expand as time goes on, since more duration may appear as the system state evolves. In addition, a corresponding vector CP_i–that represents the occurrence probabilities of each possible timing–is generated at the same time. The algorithm to calculate timing is illustrated in Algorithm 2.

Algorithm 2 takes three inputs (transition matrix P, timing distribution vector C and its corresponding probability vector CP) and produces two outputs (new timing distribution vector $C2$ and its corresponding probability vector $CP2$). We can see that timing information is associated with the Markov chain model transition matrix P. All values of elements in P are examined: if not 0, this means that the system will change its state, and the timing vectors will be used to compute the timing associated with the transition. There are 2 cases for timing calculation, as seen below

Line 7:. If the element of transition matrix P to be examined is on the diagonal, then it is $p_{j\to i}$, where $j = i$, i.e. the state does not change and the access is a cache hit. In this case, timing vectors will be expanded: the duration for a cache hit (e.g. 1 cycle) is added to all elements to timing vector C_j which keeps the timing of state s_j and the result is integrated into timing vector C_i for s_i. The corresponding probability vector CP_j is directly integrated for CP_i, since a cache hit forces the transition probability $p_{i\to i} = 1$.

Line 10:. If the transition matrix element is not on the diagonal, then it is $p_{j\to i}$, where $j \neq i$. It means the state has changed from state s_j to s_i and therefore it is a cache miss. In this case, the timing vectors will be expanded the same way as in the previous case: the duration for a cache miss (e.g. 100 cycles) is added to C_j and integrated into C_i. However, its corresponding probability CP_j is multiplied by

ALGORITHM 2: Calculate timing distribution

Data: Transition matrix P, timing distribution C, corresponding prob. CP
Result: New timing distribution $C2$, prob. $CP2$

```
N_h ← cycle number for cache hit;
N_m ← cycle number for cache miss;
C ← ∅;
CP2 ← ∅;
for c_k^i ∈ C_i and c_k^j ∈ C_j do
    if p_{j→i} ≠ 0 then
        if i=j then
            //cache hit
            c_k^i = c_k^i + N_h;
        else
            //cache miss
            c_k^i = c_k^j + N_m;
            Pr(c_k^i) = Pr(c_k^j) · p_{j→i};
        end
    end
end
C2 = C2 + C;
CP2 = CP2 + CP;
//merge timings with the same cycle number
Merge C and CP;
```

the transition probability $p_{j\to i}$. The probability result considering each transition is integrated into CP_i.

With the Algorithm 2, new timing vectors are generated based on old ones. However, in the new timing vectors there may be duplicate duration. Such mutually exclusive cases should be merged: timings with the same duration are merged by adding their probabilities. For example, there is a pair of timing and probability vectors

$$C_i = [100, 100, 201], CP_i = [0.2, 0.7, 0.1]$$

can be merged as

$$C_i = [100, 201], CP_i = [0.9, 0.1].$$

4.5. Analysis Framework

By computing timing vectors together with the state space model, we can obtain the required timing information. The framework of our computation is displayed in Figure 4.

In this framework, a transition matrix P_{n-1} is computed at every step. With the transition matrix P_{n-1}, the new state S_n is obtained using a Markov chain matrix model, which will be used for the transition matrix calculation for the next step. Meanwhile, timing vectors C_i, CP_i are generated using the transition matrix. These are fed back to the next step. Finally, timing vectors are accumulated to form the timing exceedance function.

5. ADAPTIVE METHOD

In previous sections, we have demonstrated how to use a Markov chain based model to do timing analysis. The result of this method is accurate, but since the number of states increases polynomially with a large exponent, this method is intractable. As a

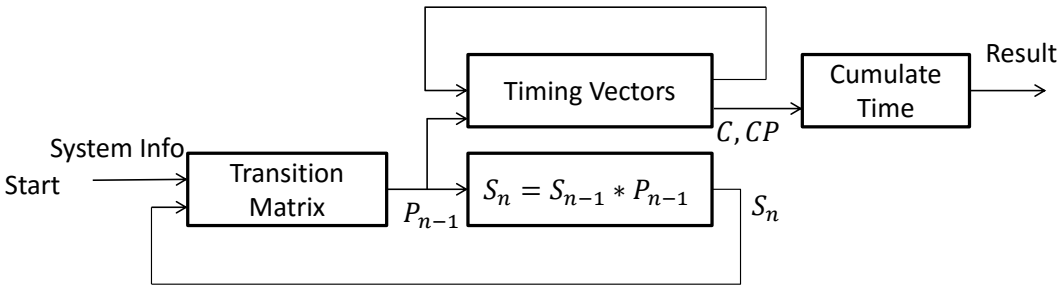


Fig. 4: Markov chain framework

result, we introduce an adaptive method to limit the number of states and to produce a result with reasonable accuracy.

5.1. State Modification

There are different ways to select some memory addresses for Markov chain states. Here we propose an adaptive method and it replaces states in the markov chain continuously.

We have shown that state s_i is used to represent a unique memory layout of the system. Suppose there are $|M|$ memory addresses, to reduce computational complexity, we would like to use only n ($n < |M|$) memory addresses to represent states. This is realized using state modification in two steps:

— State construction: for the first n addresses, we construct the state space using the Markov chain method as $\{s_0, s_1, ...\}$. This way, the number of states does not increase polynomially any more- it is limited to the value given by Equation (3). When another new memory address comes, we modify the memory addresses in state s_i in next step, instead of increasing the number of states.
— State modification: when memory address a is accessed, we first check if it is already in state s_i. If $a \in s_i$, it means this memory address is already in our state space. In this case, we do not need to modify the state. If $a \notin s_i$, it means this is a new address, and we would like to modify our state such that this address a is included in the state space. Meanwhile, one memory address in the state space will be discarded to have the same number of address. To modify the state, we first find the memory address b that is not used in future, or whose next access takes the most time steps in all memory addresses consisting the state space when it is used. This method tries to keep all memory addresses that will be used shortly and discard those memory addresses that will be used after a long time. The state s_i containing b is then removed. The new memory address a is applied to the Markov chain model and new state s_j is constructed, i.e. $s_j = s_i \setminus \{b\} \cup \{a\}$. This way, the number of states remains the same, but the states represent different memory addresses: those states s_i containing address a have been replaced by states s_j containing address b.

We can see that we select a fixed number of addresses, which results in a fixed number of states. Besides, by looking into the future memory requests, we can find which addresses will be used shortly. Hence we are able to discard those addresses that will be used at a later time, while pessimistically merging their timing information into the existing states as described in Section 5.2.

5.2. Timing Analysis

When states are changed, we need to take timing analysis into account as well, because each state is assigned different timing distributions. To obtain the safety bound of pWCET, we use a conservative method. We need to deal with the following variables for timing analysis: state occurrence vector S, timing distribution vector C and its occurrence probability vector CP.

Suppose s_i is the state containing the memory address m_d to be discarded, and s_k is the state containing all other memory addresses in s_i except the address b, i.e. $s_k = s_i \setminus \{m_d\}$. In previous section, we see that when a new address is accessed, we may remove the state s_i. Therefore the state vector which represents its occurrence probability must be modified accordingly. In the new state vector S, we use this formula to modify it:

$$Pr(s_k) = Pr(s_i) + Pr(s_k) \tag{8}$$

Let m_i be the incoming address. For the new state $s_j = s_i \setminus \{m_d\} \cup \{m_i\}$, we have $Pr(s_j) = 0$.

Apart from the state vector modification, we need to modify the timing distribution vector C and its occurrence probability vector CP. This is performed in a similar way to the modification of the state occurrence vector S. Suppose C_i and CP_i are vectors for s_i, and C_k and CP_k are vectors for s_k. Then the modification is performed using formulae

$$C_k = C_k + C_i, CP_k = CP_i + CP_k \tag{9}$$

The elements with the same number of cycles are then merged, as shown in Line 20 of Algorithm 2.

After the modifications of the state occurrence vector S, timing distribution vector C and its occurrence probability vector CP, we have transformed our Markov chain model into a new one. The next address to be accessed is applied to this new Markov chain model by using Algorithm 1 and Algorithm 2 and timing analysis can thus be performed.

5.3. Safety of the Adaptive Method

We adaptively modify states using Equation (8) and (9), which can provide pessimistic and safe results. Hereby we explain why results are safe. Let s_n be a state without state modifications from the adaptive method, i.e. a state that contains all memory addresses. Let s_a be a state with adaptive method. Note that some memory addresses may be discarded by state modifications. Thus we have $s_a \subseteq s_n$. To study if the adaptive method using s_a produces safe results compared to the method using s_n, we need to compare C_a and CP_a with C_n and CP_n. Let m_i be the incoming address and we know that $C_a = C_n$ and $CP_a = CP_n$ before accessing m_i. We need to consider following cases:

— $m_i \in s_a$: it implies that $m_i \in s_n$. Consequently there is a cache hit for both s_a and s_n. From Algorithm 2, we can compute that $\forall p, C_a[p] = C_a[p] + N_h$, where $C_a[p]$ in the right-hand side is the element in C_a before accessing m_i, $C_a[p]$ in the left-hand side is the element in C_a after accessing m_i and N_h is the number of cycles for a cache hit. This is the same for C_n. Therefore after accessing m_i, we still have $C_a = C_n$ and $CP_a = CP_n$.

— $m_i \notin s_a$ and $m_i \in s_n$: for s_a, it is a cache miss; for s_n, it is a cache hit. In the case of a cache miss, a state s_a may become a state s_i, with associated C_i and CP_i. Using Algorithm 2, we have $\forall p, C_i[p] = C_a[p] + N_m$, where N_m is the number of cycles for a cache miss.

When a cache miss happens, we need to take into account of all new states and all states use the same C_i. We sum up the probability vector elements of all states.

$CP_a[p] = \sum_i CP_a[p] \cdot p_{a \to i} = CP_a[p] \sum_i p_{a \to i} = CP_a[p]$. We can see that for a cache miss, the sum of probability vectors of new states is $CP_a = CP_n$. However, the element in C_a is larger than the element in C_n, because $N_m > N_h$. Therefore the result using s_a is safe and more pessimistic compared to the result using s_n.

— $m_i \notin s_a$ and $m_i \notin s_n$: we still have $C_a = C_n$ and $CP_a = CP_n$, but the element in C_a is added by N_m instead of N_h.

From previous discussion, we conclude that our method can provide a safe and pessimistic result. In our method, we have selected a memory address and have replaced it with incoming memory address.This address is selected to keep as much information as possible for the system: for future memory accesses, only the cache hits which are related to discarded memory addresses are ignored. Using this method, we take account of temporal characteristics of applications and make our model adaptive better to their dynamic changes, such as the case that cache locality changes during execution in terms of both cache contents and the active line number. In Section 7, we can see that this method increases result accuracy and reduces computational cost. Example 5.1 presents how to adaptively modify state space $\mathbb{S}$, state occurrence vector S, timing distribution vector C and the corresponding probability vector CP.

Example 5.1. Suppose that the memory accesses are a, b, c, a, c, and they are accessed from step 1 to step 5. We limit the distinct memory address number $n = 2$. Besides, we assume that the cache associativity $N_w = 4$, the cache hit cycle $N_h = 1$, and the cache miss cycle $N_m = 100$.

At step 1, before we access a, we construct the state space as $\mathbb{S} = \{s_0 = \emptyset, s_1 = \{a\}, s_2 = \{b\}, s_3 = \{a, b\}\}$. The state occurrence vector $S = [1, 0, 0, 0]$, timing distribution vector $C = [\emptyset, \emptyset, \emptyset, \emptyset]$, and the corresponding probability vector $CP = [\emptyset, \emptyset, \emptyset, \emptyset]$.

At step 3, we need to modify $\mathbb{S}$, S, C and CP. Before the modification, we have $\mathbb{S} = \{s_0 = \emptyset, s_1 = \{a\}, s_2 = \{b\}, s_3 = \{a, b\}\}$, with $S = [0, 0, 0.25, 0.75]$, $C = [\emptyset, \emptyset, [200], [200]]$ and $CP = [\emptyset, \emptyset, [0.25], [0.75]]$.

Then we change the state space to $\mathbb{S} = \{s_0 = \emptyset, s_1 = \{a\}, s_2 = \{c\}, s_3 = \{a, c\}\}$. b is replaced by c, because it is not used in following accesses. The associated vectors are changed accordingly: $S = [0.25, 0.75, 0, 0]$, $C = [[200], [200], \emptyset, \emptyset]$ and $CP = [0.25, 0.75, \emptyset, \emptyset]$. We can see that after state modification, we have $Pr(s_1 = \{a\}) = 0.75$, which is a pessimistic case of $Pr(s_1 = \{a, b\}) = 0.75$. In the same way, we change the state from $Pr\{b\} = 0.25$ to $Pr\{\emptyset\} = 0.25$. The corresponding C and CP are modified accordingly.

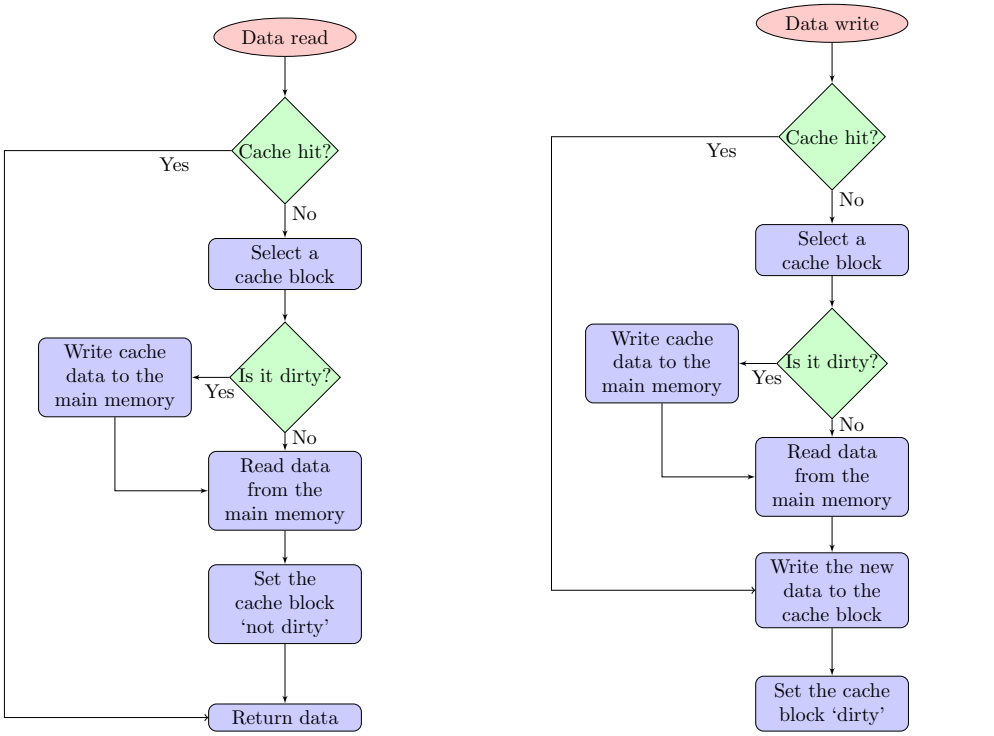
6. EXTENSION TO DATA CACHES

We have demonstrated our Markov chain approach that can be applied to instruction caches directly. However, writing policies make data caches behave differently from instruction caches. In this section, we explain how to extend our approach to data caches.

6.1. Data Cache Writing Policies

There exist two writing policies for data caches: write-through and write-back policies. A write-through policy writes data to both the cache and the main memory at the same time. This can be modeled very easily and our Markov chain method can be applied in a straightforward fashion.

However, a write-back data cache behaves differently and it is often combined with a write-allocate policy, as illustrated in Figure 5. When reading or writing data from a write-back cache, we need to check if the selected cache block is set as 'dirty'. If so, the data in the cache block must be sent to the main memory first, because it has been modified, which results in an additional latency.



(a) Read data from a data cache.

(b) Write data to a data cache.

Fig. 5: Read and write for a write-back data cache with a write-allocate.

To extend our approach to data caches, we assume that when reading or writing, a cache access latency is N_h, and a main memory access latency is N_m. This is the same as what we used for instruction caches. From Figure 5, we can calculate latency L for the following scenarios:

— Cache hit: $L = N_h$.
— Cache miss and 'not dirty': $L = N_m$.
— Cache miss and 'dirty': $L = 2N_m$.

6.2. Method Modification

Compared to instruction caches, there is an additional latency N_m for cache misses in the presence of 'dirty' cache blocks. Thus we need to modify Algorithm 1 and 2. We introduce binary variables $B_d^a, B_t^a \in \{true, false\}$ to represent if the cache block with address a is dirty and the type of the data access to a, respectively. When the block with address a is dirty, $B_d^a = true$; otherwise $B_d^a = false$. If it is a data write, $B_t^a = true$; otherwise $B_t^a = false$.

First, we add additional operations to Algorithm 1 as follows:

Line 5:. In the case of a cache miss, we set the block as 'not dirty' if it is a data read, i.e. if $B_t^a = false, B_d^a = false$.
Line 15:. We set the block as 'dirty' if it is a data write, i.e. if $B_t^a = true, B_d^a = true$.

Next we modify Algorithm 2 to account for additional latencies due to dirty blocks as follows:

Line 7:. We modify the case of a cache hit, such that if a block is not dirty, the latency is N_h; otherwise it is N_m. Then we remove Line 9 and add the following: if $B^a_d = false, c^i_k = c_k + N_h$; otherwise $c^i_k = c^i_k + N_m$.
Line 10:. We modify the case of a cache miss in a similar way. If a block is not dirty, the latency is N_m; otherwise it is $2N_m$. Note that when using the adaptive method, some addresses may be discarded, which results in empty cache blocks. We use $B^a_{discard} = true$ to indicate the address a has been discarded before, and $B^a_{discard} = false$ otherwise. When we access discarded addresses in future, we assume pessimistically that the blocks with such addresses are dirty. Consequently, we remove Line 12 and add the following: if $B^a_d = false \wedge B^a_{discard} = false, c^i_k = c^j_k + N_m$; otherwise $c^i_k = c^j_k + 2N_m$.

By adding operations related to dirty blocks to Algorithm 1, we are able to tell if each cache block is dirty or not. We modify Algorithm 2 to account for additional latencies caused by dirty blocks, and use pessimistic assumptions while applying the adaptive method.

7. BENCHMARKS EVALUATION

In this section, we evaluate our methodology using real-world benchmark applications. We chose the Mälardalen benchmarks [Gustafsson et al. 2010], a popular benchmark suite used for WCET evaluation and analysis. We perform SPTA using our adaptive Markov chain model, and compare its results with results from another state-of-the-art SPTA methodology. All benchmarks are performed with a dual-core Intel Duo CPU running at 3.0 GHz with 4GB memory.

Our experiments use the SoCLib open platform[2] to simulate our design under test. SoCLib supports several processor architectures: we adopt the MIPS 32-bit processor architecture. The Mälardalen benchmark suite was compiled into MIPS ISA with the Sourcery CodeBench tool from Mentor Graphics[3]. The platform is equipped with a single MIPS processor with an L1 instruction cache, which has been modified to use evict-on-miss random replacement policy. Our experiments are performed for an instruction cache and a write-back, write-allocate cache.

For industrial and avionic embedded systems, cache associativity is usually fairly small. For example, the LEON3[4] processor has a configurable cache between 1 and 4 ways. Thus we set the cache size as 512 bytes, with 4-way associativity and 4-byte cache block. For each cache miss, we assume a duration of 100 cycles and thus a dirty data cache block causes another 100 cycles; for each cache hit, the delay is 1 cycle. Memory address traces are generated by the platform, which are used for SPTA and adaptive Markov chain model analysis. We considered modulo placement only.

Benchmarks[5] used for analysis are listed in Table I. We select the benchmarks that do not require hard floating point unit that is absent in our SoCLib platform.

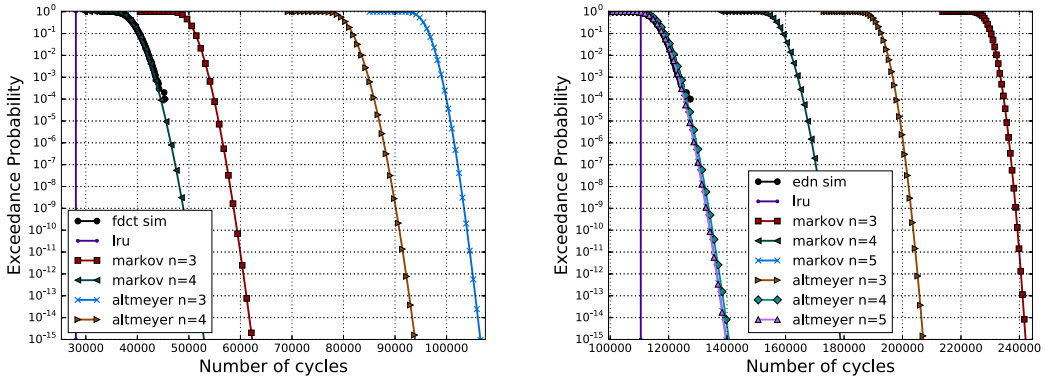
7.1. Model Accuracy

In this section, results from the adaptive Markov chain model are compared with simulations to verify its accuracy. For each cache set, different number of memory addresses are selected to see their impact on timing analysis. We select 10^{-15} as the exceedance probability of interest, since the maximum allowed failure rate is 10^{-9} per hour for commercial airborne systems, which is equivalent to an exceedance probabil-

[2]http://www.soclib.fr/
[3]http://www.mentor.com/embedded-software/sourcery-tools/
[4]http://www.gaisler.com/index.php/products/processors/leon3
[5]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Benchmark	Description
expint	Series expansion for computing an exponential integral function
bs	Binary search
duff	Unstructured loop with known bound
statemate	Automatically generated code
fdct	Fast discrete cosine transform
jfdctint	Discrete cosine transformation
ndes	Bit manipulation, shifts, array and matrix calculations
compress	Data compression
edn	Vector multiplication and array handling
adpcm	Adaptive pulse code modulation
bsort100	Bubble sort
matmult	Matrix multiplication
fir	Finite impulse response filter

Table I: Benchmarks



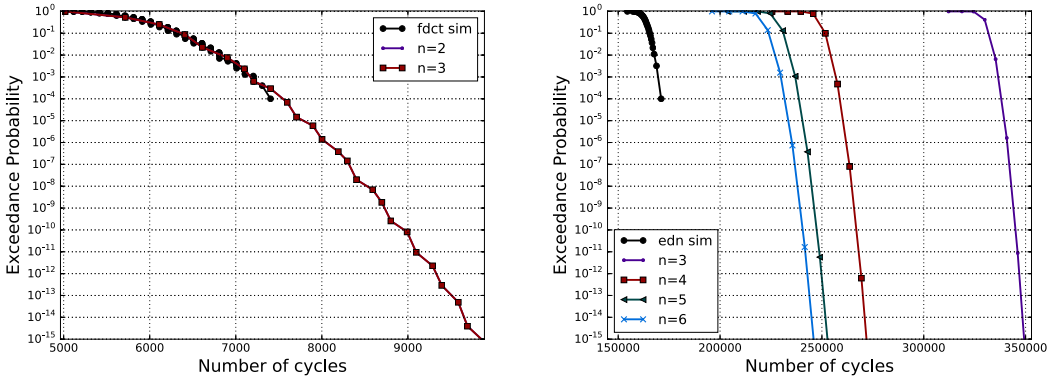
(a) Benchmark fdct. Total memory access: 1632. Distinct memory access: 267.

(b) Benchmark edn. Total memory access: 2398. Distinct memory access: 417.

Fig. 6: Adaptive Markov chain model accuracy using instruction caches. A varying number of memory addresses are used in adaptive Markov chain model for comparison with simulations.

ity of around 10^{-13} [Cucu-Grosjean et al. 2012]. Thus we estimate the time at 10^{-15} as a conservative result.

For the sake of space limitations, Figure 6 shows comparisons between simulations and a subset of the benchmarks using instruction caches and Figure 7 shows comparisons using data caches. Figure 6a and Figure 7a display the comparison between simulations and FDCT benchmark; Figure 6b and Figure 7b show the comparison for EDN benchmark. We ran 10,000 simulations to sample the timing behavior of the benchmark and performed cache analyses with the memory traces using our proposed approach. Simulated time is obtained for each simulation. On each figure, the x-axis shows the number of cycles and y-axis represents the exceedance probability for corresponding cycles. This is called probabilistic WCET (pWCET), because for each WCET estimate, there is an associated exceedance probability. When we compare different results, we can see the WCET estimate for a specific exceedance probability.



(a) Benchmark fdct. Total memory access: 364. Distinct memory access: 47.

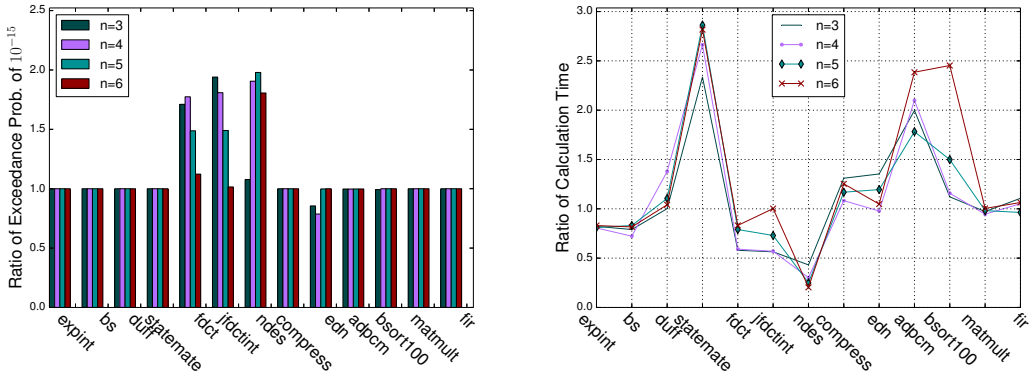
(b) Benchmark edn. Total memory access: 11551. Distinct memory access: 524.

Fig. 7: Adaptive Markov chain model accuracy using write-back data caches with write-allocate. Different number of memory addresses are used in adaptive Markov chain model for comparison with simulations.

We can see as we increase the number of memory addresses n, the result from the adaptive Markov chain comes closer to that from simulations. Take Figure 6a for example, for $n = 3$ the number of cycles is 62,000 at the exceedance probability of 10^{-15}; for $n = 4$ it becomes 53,000, reducing the estimate pessimism by using more memory addresses. They will eventually produce the same result if all memory addresses are applied to the Markov chain method, and increasing the number n does not change the result anymore, as illustrated for $n = 4$ in Figure 6a and $n = 2, 3$ in Figure 7a. In general, both the Markov chain methodology and simulations match well. Since simulations are performed randomly, there is a variance for each simulation, but the difference is not significant. However, as the probability goes down, fewer simulation samples are available, to the point where the results are not reliable: at the tail of the simulation plot, there is an obvious deviation between simulations and Markov chain methodology. This deviation is due to the lack of simulation samples, which were limited to constrain the simulation to feasible times. As the number of simulation samples increases, the simulation result converges to the result of our method. We thus conclude that our method can perform timing analysis accurately.

7.2. Comparison with Altmeyer SPTA

In this section, we use instruction caches and compare the results from our methodology with that from the state-of-the-art SPTA proposed by [Altmeyer et al. 2015], which is referred as “Altmeyer SPTA“. In “Altmeyer SPTA“, memory addresses are divided into two independent parts: the state enumeration part and cache contention part. The addresses in the state enumeration part are used for a detailed analysis. A cache state is represented as a triple $CS = (E, P, D)$, where E contains memory addresses for the state, P is the probability of the state and D is the miss distribution. Every time an address is accessed, the update function is applied to update cache state. If the memory is not in the state enumeration part, update function evicts memory address in E; otherwise, the memory address is put into E. The probability P and miss distribution D are modified accordingly by the update function. By detailed analysis, a timing distribution can be generated.



(a) Accuracy comparison.

(b) Time comparison.

Fig. 8: Comparison with state-of-the-art SPTA. Accuracy and calculation time are compared respectively using different number of memory addresses.

To perform timing analysis for cache contention part, the notion of cache contention is introduced. All memory accesses are regarded as independent, and the lower bound probability of each memory access is calculated. Cache contention is used to see how memory addresses content for cache blocks. A simulation S is used to represent potentially conflicting addresses. If the accessed memory is not in S, the hit probability is 0. Otherwise, it is calculated using reuse distance, stack distance and cache associativity. This way, hit probabilities for all memory accesses are calculated and the timing distribution can be obtained by convolution. Since this part is independent of detailed analysis of state enumeration part, the convolution of timing distributions is the final distribution of the program.

The Altmeyer SPTA consists of two parts and the enumeration part uses the most-used memory addresses. This is different from our method, in which we use only one part, and we change the memory addresses in the state space adaptively.

In Figure 8, we compare the calculation accuracy and time of two different methods. The number of memory addresses for each cache set n ranges from $n = 3$ to $n = 6$. We start by using a small number of different memory addresses for each cache set $n = 3$. With such a number, some benchmarks show similar results to simulations, while others exhibit conservative timing predictions. As we increase memory addresses up to $n = 6$, most benchmarks have reached a point where further memory address increment improves the result accuracy slowly. The memory trace file sizes of the benchmarks are in ascending order from left to right in Figure 8.

Figure 8a shows the cycle number ratio between Altmeyer SPTA and the adaptive Markov chain based method. At the exceedance probability of 10^{-15}, we obtain estimated number of cycles using Altmeyer SPTA and the adaptive Markov chain model and they are represented as N_a and N_m, respectively. We calculate the cycle number ratio R_c as

$$R_c = \frac{N_a}{N_m}$$

We can see that when $R_c > 1$, the Altmeyer SPTA is more pessimistic; when $R_c < 1$, the adaptive Markov chain model is more pessimistic. Otherwise both methods pro-

duce the same result. On average, the geometric mean of Altmeyer SPTA estimates 11% more cycles than our adaptive Markov chain based method.

Figure 8b represents the time ratio between the Altmeyer SPTA and adaptive Markov chain based method. We use T_a to denote the calculation time for Altmeyer SPTA and T_m for the adaptive Markov chain model. The time ratio R_t is computed as

$$R_t = \frac{T_a}{T_m}$$

Figure 8b illustrates that the Altmeyer SPTA takes the approximately the same amount of time as our Markov model, with Altmeyer SPTA being 1% slower on average. The calculation time ratio varies within a limited range for all benchmarks (from 0.2 to 2.9), and overall the time difference between two methods is not statistically significant according to t-test at 95% confidence (p=0.22).

7.3. Comparison with LRU

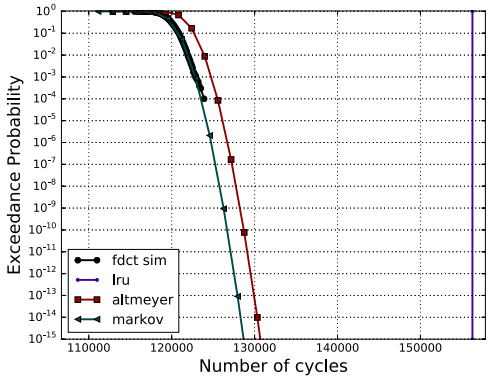
In this section, we study impacts of cache on LRU replacement policy and random replacement policy. In Figure 9, we use *fdct* and apply different cache sizes and associativities. For a single-path program, the number of cycles is constant for caches with LRU policy. We use a simulation to obtain the number of cycles using LRU policy and plot it as a vertical line.

We can see that in Figure 9a and Figure 9b, LRU performs worse than random replacement, i.e. the number of cycles from LRU policy is larger than that using random replacement policy. This is because for a smaller cache size, there are fewer cache sets. As a result, there are more memory accesses for each cache set. When the number of accesses becomes larger, the LRU performance becomes worse, since there are more opportunities to replace a cache block before its future use. Random replacement policy, however, is not affected so significantly. Each cache block is replaced randomly, which makes it possible to keep any cache block for future use. In the worst case, pathological case may occur for LRU caches, i.e. there are always cache misses for memory accesses in a cache set, since too many distinct memory addresses are used in such a pattern that they are evicted before their next accesses. A time-randomized cache can avoid such pathological cases since it evicts memory blocks randomly.

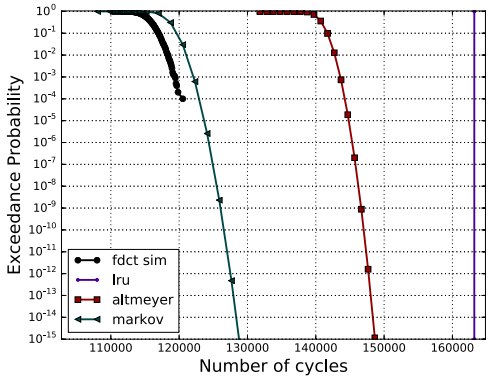
Figure 9c and Figure 9d illustrate that as the cache size increases, the number of cycles decreases significantly, especially for LRU caches. On average, a larger cache size indicates fewer memory blocks for each cache set, which may avoid pathological cases effectively for LRU caches, which reduces number of cycles dramatically.

In addition, there are no associativity constraints on the use of our approach. In Figure 9e and Figure 9f, we can see as cache associativity increases, our method can still be applied. The accuracy may be compromised, because more memory addresses are accessed when associativity increases. Some information is lost due to limited number of used blocks n, which compromises timing analysis result. The general rule is that n should be as large as possible, given the available computational resources.

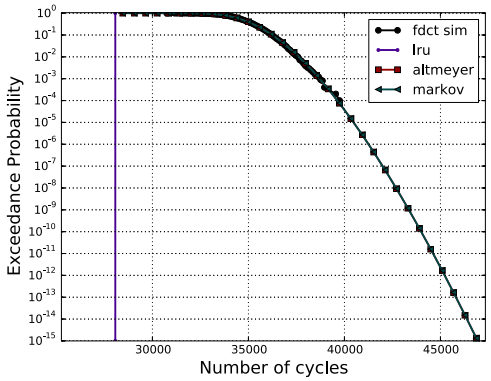
Several previous studies have done comparisons between random and LRU replacement policies [Smith and Goodman 1983; Smith and Goodman 1985; Quinones et al. 2009; Kosmidis et al. 2013a]. Our experiments show that when the code size is larger than cache size, random policy helps reduce cache misses, which confirms the conclusion from previous studies that random policy can avoid pathological cases effectively. However, note that our results depend on the code layout of benchmarks and we analyze the cache impact using a particular code layout, i.e. the trace from the platform. This is only one of the entire code layout space. If the code layout changes, different results may be produced, because we have adopted set-associative caches with modulo placement policy in the experiments. The execution times for LRU and time-



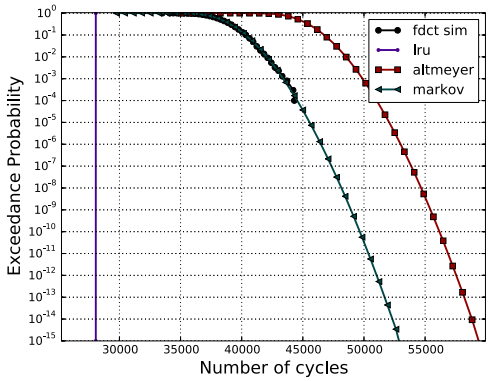
(a) Cache size: 256 bytes. Associativity: 2.



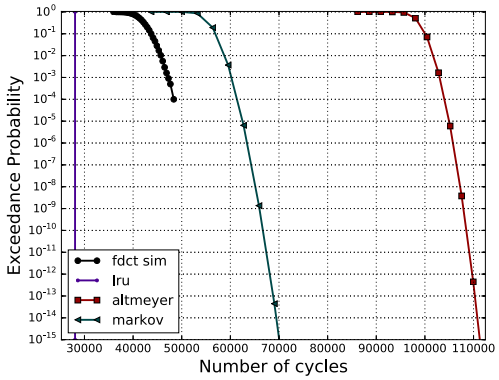
(b) Cache size: 256 bytes. Associativity: 4.



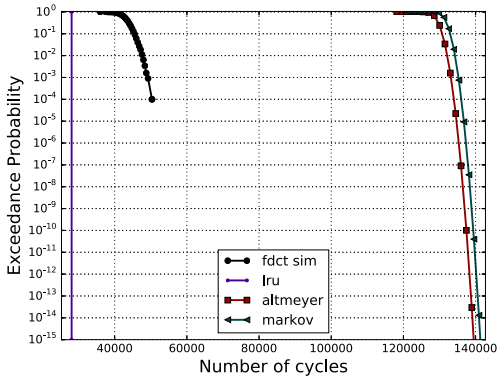
(c) Cache size: 512 bytes. Associativity: 2.



(d) Cache size: 512 bytes. Associativity: 4.



(e) Cache size: 512 bytes. Associativity 8.



(f) Cache size: 512 bytes. Associativity 16.

Fig. 9: Benchmark fdct. Comparison with LRU replacement with different cache sizes and associativities. Number of used blocks n=6.

randomized caches may be different, i.e. they may be shorter or longer compared to the presented results, depending on changes of the code layout. In this section, we do not compare average performance of random and LRU policies, since we do not have memory traces of all code layouts.

8. CONCLUSIONS

In this paper, we have demonstrated an adaptive Markov chain based Static Probabilistic Timing Analysis (SPTA) methodology. Our methodology is based on a non-homogeneous Markov chain model, which explores state space modeling for one cache set, and convolves different sets to generate final timing information. To reduce computational complexity, the state space can be limited to the specified level. The state space is modified adaptively, such that selected addresses can be replaced by new incoming addresses in the state space with good accuracy, while maintaining the same number of states. By reducing the number of addresses used for state modification, we can find a compromise between calculation accuracy and time.

Benchmark applications are used to verify accuracy of this methodology by using simulations based on SoCLib platform with MIPS processor architecture. Its results are compared to state-of-the-art SPTA methodology. It shows that with the adaptive state modification, our methodology has improved accuracy of results using less amount of calculation time. We also demonstrate how to evaluate cache impacts on system timing behaviors using the proposed method, which can help designers to select cache parameters of real-time embedded systems.

As future work, we can address several aspects: Simultaneous running of multiple programs and hybrid SPTA/MBPTA are two examples. In addition, we only explored single-path programs in this paper. However, it can be extended to multi-path programs by identifying the worst-case path. Fully extending the approach to multi-path programs is also part of our future work.

REFERENCES

J. Abella, D. Hardy, I. Puaut, E. Quinones, and F.J. Cazorla. 2014a. On the Comparison of Deterministic and Probabilistic WCET Estimation Techniques. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. 266–275. DOI:http://dx.doi.org/10.1109/ECRTS.2014.16

J. Abella, E. Quinones, F. Wartel, T. Vardanega, and F.J. Cazorla. 2014b. Heart of Gold: Making the Improbable Happen to Increase Confidence in MBPTA. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. 255–265. DOI:http://dx.doi.org/10.1109/ECRTS.2014.33

Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. 2004. Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference (ACM-SE 42)*. ACM, New York, NY, USA, 267–272. DOI:http://dx.doi.org/10.1145/986537.986601

Sebastian Altmeyer, Liliana Cucu-Grosjean, and RobertI. Davis. 2015. Static probabilistic timing analysis for real-time systems using random replacement caches. *Real-Time Systems* 51, 1 (2015), 77–123. DOI:http://dx.doi.org/10.1007/s11241-014-9218-4

S. Altmeyer and R.I Davis. 2014. On the correctness, optimality and precision of Static Probabilistic Timing Analysis. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. 1–6. DOI:http://dx.doi.org/10.7873/DATE.2014.039

Jan Beirlant, Yuri Goegebeur, Johan Segers, and Jozef Teugels. 2006. *Statistics of Extremes: Theory and Applications*. John Wiley & Sons.

Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. *SIGPLAN Not.* 41, 6 (June 2006), 158–168. DOI:http://dx.doi.org/10.1145/1133255.1134000

P. Bernardara, F. Mazas, X. Kergadallan, and L. Hamm. 2014. A two-step framework for over-threshold modelling of environmental extremes. *Natural Hazards and Earth System Science* 14, 3 (2014), 635–647. DOI:http://dx.doi.org/10.5194/nhess-14-635-2014

Guillem Bernat, Alan Burns, and Martin Newby. 2005. Probabilistic Timing Analysis: An Approach Using Copulas. *J. Embedded Comput.* 1, 2 (April 2005), 179–194. http://dl.acm.org/citation.cfm?id=1233760.1233763

G. Bernat, A Colin, and S.M. Petters. 2002. WCET analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*. 279–288. DOI:http://dx.doi.org/10.1109/REAL.2002.1181582

Guillem Bernat, Antoine Colin, and Stefan Petters. 2003. pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems. *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS* (2003).

A. Burns and S. Edgar. 2000. Predicting computation time for advanced processor architectures. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*. 89–96. DOI:http://dx.doi.org/10.1109/EMRTS.2000.853996

Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. 2013. PROARTIS: Probabilistically Analyzable Real-Time Systems. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article 94 (May 2013), 26 pages. DOI:http://dx.doi.org/10.1145/2465787.2465796

C. Chen, L. Santinelli, J. Hugues, and G. Beltrame. 2016. Static probabilistic timing analysis in presence of faults. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. 1–10. DOI:http://dx.doi.org/10.1109/SIES.2016.7509422

L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F.J. Cazorla. 2012. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. 91–101. DOI:http://dx.doi.org/10.1109/ECRTS.2012.31

Rob Davis. 2013. Improvements to static probabilistic timing analysis for systems with random cache replacement policies. *RTSOPS 2013* (2013), 22–24.

R.I Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. 2013. Analysis of Probabilistic Cache Related Pre-emption Delays. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. 168–179. DOI:http://dx.doi.org/10.1109/ECRTS.2013.27

Laurens De Haan and Ana Ferreira. 2007. *Extreme value theory: an introduction*. Springer.

S. Edgar and A. Burns. 2001. Statistical analysis of WCET for scheduling. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*. 215–224. DOI:http://dx.doi.org/10.1109/REAL.2001.990614

David Griffin and Alan Burns. 2010. Realism in Statistical Analysis of Worst Case Execution Times.. In *WCET*. 44–53.

David Griffin, Benjamin Lesage, Alan Burns, and Robert I. Davis. 2014. Static Probabilistic Timing Analysis of Random Replacement Caches Using Lossy Compression. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems (RTNS '14)*. ACM, New York, NY, USA, Article 289, 10 pages. DOI:http://dx.doi.org/10.1145/2659787.2659809

Emil Julius Gumbel. 2012. *Statistics of extremes*. Courier Corporation.

J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010) (OpenAccess Series in Informatics (OASIcs))*, Vol. 15. 136–146. DOI:http://dx.doi.org/10.4230/OASIcs.WCET.2010.136

Jeffery Hansen, Scott A Hissam, and Gabriel A Moreno. 2009. Statistical-based wcet estimation and validation. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*.

Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. 2013a. A Cache Design for Probabilistically Analysable Real-time Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*. EDA Consortium, San Jose, CA, USA, 513–518. http://dl.acm.org/citation.cfm?id=2485288.2485416

Leonidas Kosmidis, Charlie Curtsinger, Eduardo Quiñones, Jaume Abella, Emery Berger, and Francisco J. Cazorla. 2013b. Probabilistic Timing Analysis on Conventional Cache Designs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*. EDA Consortium, San Jose, CA, USA, 603–606. http://dl.acm.org/citation.cfm?id=2485288.2485435

L. Kosmidis, E. Quinones, J. Abella, T. Vardanega, I. Broster, and F.J. Cazorla. 2014a. Measurement-Based Probabilistic Timing Analysis and Its Impact on Processor Architecture. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*. 401–410. DOI:http://dx.doi.org/10.1109/DSD.2014.50

L. Kosmidis, E. Quiones, J. Abella, G. Farrall, F. Wartel, and F. J. Cazorla. 2014b. Containing timing-related certification cost in automotive systems deploying complex hardware. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. DOI:http://dx.doi.org/10.1145/2593069.2593112

L. Kosmidis, R. Vargas, D. Morales, E. Quiones, J. Abella, and F. J. Cazorla. 2016. TASA: Toolchain-Agnostic Static Software randomisation for critical real-time systems. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. DOI:http://dx.doi.org/10.1145/2966986.2967078

B. Lesage, D. Griffin, S. Altmeyer, and R.I. Davis. 2015a. Static Probabilistic Timing Analysis for Multi-path Programs. In *Real-Time Systems Symposium, 2015 IEEE*. 361–372. DOI:http://dx.doi.org/10.1109/RTSS.2015.41

Benjamin Lesage, David Griffin, Frank Soboczenski, Iain Bate, and Robert I. Davis. 2015b. A Framework for the Evaluation of Measurement-based Timing Analyses. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS '15)*. ACM, New York, NY, USA, 35–44. DOI:http://dx.doi.org/10.1145/2834848.2834858

Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. 2011. A New Way About Using Statistical Analysis of Worst-case Execution Times. *SIGBED Rev.* 8, 3 (Sept. 2011), 11–14. DOI:http://dx.doi.org/10.1145/2038617.2038619

Grant Martin. 2006. Overview of the MPSoC Design Challenge. In *Proceedings of the 43rd Annual Design Automation Conference (DAC '06)*. ACM, New York, NY, USA, 274–279. DOI:http://dx.doi.org/10.1145/1146909.1146980

Enrico Mezzetti, Marco Ziccardi, Tullio Vardanega, Jaume Abella, Eduardo Quiñones, and Francisco J Cazorla. 2015. Randomized Caches Can Be Pretty Useful to Hard Real-Time Systems. *Leibniz Transactions on Embedded Systems* 2, 1 (2015), 01–1.

E. Quinones, E.D. Berger, G. Bernat, and F.J. Cazorla. 2009. Using Randomized Caches in Probabilistic Real-Time Systems. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*. 129–138. DOI:http://dx.doi.org/10.1109/ECRTS.2009.30

Jan Reineke. 2014. Randomized Caches Considered Harmful in Hard Real-Time Systems. *Leibniz Transactions on Embedded Systems* 1, 1 (2014), 03–1–03:13. http://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v001-i001-a003

Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing predictability of cache replacement policies. *Real-Time Systems* 37, 2 (2007), 99–122. DOI:http://dx.doi.org/10.1007/s11241-007-9032-3

Michael Schlansker, Robert Shaw, and Sivaram Sivaramakrishnan. 1993. *Randomization and associativity in the design of placement-insensitive caches*. Hewlett-Packard Laboratories.

Richard Serfozo. 2009. *Basics of applied stochastic processes*. Springer.

James E. Smith and James R. Goodman. 1983. A Study of Instruction Cache Organizations and Replacement Policies. In *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA '83)*. ACM, New York, NY, USA, 132–137. DOI:http://dx.doi.org/10.1145/800046.801648

James E. Smith and James R. Goodman. 1985. Instruction cache replacement policies and organizations. *IEEE Trans. Comput.* 34, 3 (1985), 234–241.

N. Topham and A Gonzalez. 1999. Randomized cache placement for eliminating conflicts. *Computers, IEEE Transactions on* 48, 2 (Feb 1999), 185–192. DOI:http://dx.doi.org/10.1109/12.752660

F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F.J. Cazorla. 2013. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*. 241–248. DOI:http://dx.doi.org/10.1109/SIES.2013.6601497

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages. DOI:http://dx.doi.org/10.1145/1347375.1347389

Shuchang Zhou. 2010. An Efficient Simulation Algorithm for Cache of Random Replacement Policy. In *Network and Parallel Computing*, Chen Ding, Zhiyuan Shao, and Ran Zheng (Eds.). Lecture Notes in Computer Science, Vol. 6289. Springer Berlin Heidelberg, 144–154. DOI:http://dx.doi.org/10.1007/978-3-642-15672-4_13

Received ; revised ; accepted